MIPP Offline Software Users Manual

Version 0.2

Compiled on March 16, 2011

Contents

Chapter 1

Overview

This chapter is intended to give the user a quick introduction to the broad features of the MIPP software. Not too many how-to details but the big picture. The main web page for the MIPP offline is here:

> `http://ppd.fnal.gov/experiments/e907/OfflineSoftware/`

and should be the first place you look for information. If you don't find what you're looking for, complain and we'll add it.

If you start from scratch with the MIPP software, you'll probably want to make yourself a test code release using `SRT`. You'll then probably want to add a package to the release or check out an existing package from the MIPP CVS repository. You'll edit the `C++` code you find there, maybe booking some histograms and filling them using the `ROOT` package. When you're done you'll commit your changes back to the `CVS` repository.

1.1 Programming Tools

Here I'll list the basic tools one needs to have at least a passing understanding of to work with the MIPP software.

1.1.1 C++

The MIPP software is mostly written in `C++` although there is still some Fortran floating around the edges. In general, new software written for MIPP should be written in C++. This manual cannot teach you C++, but you can get an idea of what C++ looks like by browsing the code:

```
http://enrico1.physics.indiana.edu/mipp/doxygen/
                         offline/html/files.html
```

and by reading the coding conventions at:

```
http://ppd.fnal.gov/experiments/e907/OfflineSoftware/
                         CodingConventions/
```

The coding conventions were gleaned from several sources and provide some order to the large variations in syntax that C++ allows and include some "best practices" tips.

1.1.2 STL: `http://www.sgi.com/tech/stl/`

`STL` stands for the "standard template library". While it is technically an extension of C++, it is now a standard part of the language. STL provides software for managing things like vectors, lists, maps, as well as standard searching and sorting routines. A must-know for a `C++` programmer.

1.1.3 ROOT: `http://root.cern.ch/`

`ROOT` is a large software tool kit provided by Rene Brun et al. and is roughly the `C++` equivalent of what CERNLIB was for FORTRAN. ROOT provides I/O, GUI's, histogramming, and analysis tools. Some of ROOT's functionality duplicates things that are in the `STL`. Since `STL` is an ANSI standard, MIPP prefers the use of `STL` before `ROOT` if one has a choice.

1.1.4 PostgreSQL, XML, and all that

These are packages for managing input data from various sources. *PostgreSQL* handles queries to a database. *XML* provides a well defined format for exchanging text-formatted data based on the HTML markup language. Let's not worry about these just yet...

1.2 Code Management Tools

1.2.1 CVS

`CVS` is the program we use to share code, manage versions, and track changes to the code. It is a standard UNIX command (type "man cvs" at a UNIX

prompt to read all about it). The only command users typically ever need is "cvs commit" which is used to publish changes you've made in your local copy of a piece of code to the rest of the collaboration.

1.2.2 SRT

`SRT` stands for Software Release Tools. `SRT` is a Fermilab program used to manage the building of code across several platforms. `SRT` also provides hooks for linking your version of the MIPP software to a "master copy" (aka the "base release") so that you don't have to have all the MIPP software checked out to work on some small piece of it.

1.3 MIPP Offline Software Packages

To help give people who are new to the software get a feel for the lay of the land, I've listed here the packages that currently exist in the MIPP software and a short description of what they do.

1.3.1 Detector Geometry

Bfield Return values of the B field in JGG and Rosie magnets.

Geometry Answers questions like: How big is the TPC drift volume? What material is a point in? Which detector is a point located in? How far to the next boundary?

ConnectionMap Allows used to navigate through the various electronic addresses given to the electronics channels in the detector.

1.3.2 Simulation

e907mc Main GEANT3 simulation program

dd_geant and trim_d0lib Used by e907mc

CalDigitizer Simulate response of calorimeters

ChamDigitizer   Simulate response of drift chambers and multi-wire chambers

CkovDigitizer   Simulate response of threshold Čerenkov detector

RICHDigitizer   Simulate response of RICH

TOFDigitizer   Simulate response of time of flight counters

TPCDigitizer   Simulate response of TPC

E907MCInterface   Wrappers to handle FORTRAN/C++ interface

Geant3Interface   Wrappers to handle C++ interface to GEANT3

MCTruth   True MC tracks and vertices.

MCUtil   Utility programs related to MC

1.3.3   Data Format and I/O

RawData   Basic hit data from the detector

MCClasses   Detector hit and track data from the `e907mc` simulation

EventDataModel   Data format used by the offline

MippIo   Raw data format used by the DAQ

Raw2Root   Convert from raw DAQ format to offline format

IoModules   Reading and writing events from files

1.3.4   Reconstruction and Analysis

JobControl   Basic interface for plugging analysis and reconstruction code into the analysis program. Provides the *anamipp* executable and JobModule base class.

RecoBase   Base classes for reconstruction. In particular, `RBKTrack` object is defined there, which is the standard MIPP track object from which other tracking objects should inherit.

Config   Configuration package. Mostly used through JobControl.

DSTMaker and MIPPEventSummary   create data summary files using MIPPEventSummary format

RunInfo   gives access to beam momentum, target material, gas pressures, HV settings, and other information that is constant over the run.

1.3.5   Tracking

TrkRBase

TrkRBase package contains basic algorithms to reconstruct wire chamber data. In particular `TrkRBase` sequence defined in TrkRBase.xml does the following:

1. Converts BC, DC, and PWC digit information to `Wire`'s. Hot wires are dropped, BC and DC TDC info is converted to drift time. You can find a given chamber's in `Cal` data bucket, folder names are "BC1", "DC1", "PWC5", etc.

2. Groups adjacent wires into `WireClust`'s and computes location of the cluster in that chamber plane view. Clusters are also written into `Cal` data bucket, same folder names.

3. For each chamber finds all possible crosses of clusters and writes them out as `WireCross`'s into `Cal` bucket, same folder names.

4. Looks for straight-line TrackSeg's in three groups: BC's, DC1-3, and DC4+PWC's. Objects are written out into `Reco` data bucket, folders "/TrkRBaseBC", "/TrkRBaseC123", "/TrkRBaseC456".

5. Takes C123 and C456 track segments and if they match sufficiently well, combines them into a `TrkCand`'s which is a candidate track with a fit momentum

TPCReco

TPCReco takes all the TPC information, looks for tracks and does a helical fit. Include `TPCReco` sequence from TPCReco.xml.

SPFit

SPFit is simple global tracking algorithm. It takes `TPCRTrack`'s, `TrkCand`'s, `TrackSeg`'s and `WireClust`'s and combines all that information into global `SPTrk`'s (which inherit from `RBKTrack`. Include `SPTrkBuilder` module with its configuration from SPTrkBuilder.xml **after `TrkRBase`** and `TPCReco` sequences. You can then retrieve a vector of `RBKTrack`'s using the following code:

```
std::vector<const RBKTrack*> trkList;
evt.Reco().Get(''/sptrk/SPTrks'', trkList);
```

It is in your best interest to retrieve `SPTrk`'s as `RBKTrack`'s, since once a better tracking algorithm appears, your code will benefit from it when you change the folder where tracks are being retrieved from.

VertexReco

1. VtxDAFit combines Tracks (from tracking) into vertices.

2. VtxConFit takes a list of vertices (from VtxDAFit) and re-fits all tracks and the vertex position with the constraint that all tracks do originate at the vertex.

1.3.6 Particle ID

RICHReco   RICH ring reconstruction

1.3.7   Miscellaneous Packages

AEMStatsPlotter   Generates plots of run statistics for presentation at Fermilab All Experimenters' Meetings

BatchProc   Submit jobs to farms and grid

ColorScale   Used by EventDisplay

EventDisplay   Display single events in the detector. Useful during data acquisition and offline.

MippDatabase and MippDatabaseBase   Interface to PostGres DBs

MippXML   XML interface

NumericalMethods   Numerical methods

setup   Information on tagged releases, setup scripts, and setupModule

SoftRelTools   SRT builds itself

SRT_MIPP   MIPP extensions and customizations to SRT

Ziptrack   Generates magnetic field maps for use by Bfield from Ziptrack data

Chapter 2

The Event Data Structure

This chapter covers the basics about how to get from and write data to a MIPP events.

2.1 The event structure

The `EventDataModel` package defines the MIPP event structure. The "public" face to the event data users see is the class "EDMEventHandle". The event data is stored on several "branches" each representing a different stage in the data processing. Let's look at some sample code:

```
void AClass::AFunc(EDMEventHandle& evt)
{
//==============================================================
// ``AFunc'' is a method of ``AClass'' which is passed an event
// handle. The handle is passed by reference (Hence the ``&'')
//==============================================================

  // The event header contains run/event numbers, date/time,
  // trigger info. etc.
  EDMEventHeader& head = evt.Header();

  // The ``MC'' branch contains the input MC vector information
  // In the case of real data it is empty
  EDMDataBucket&  mc = evt.MC();

  // The ``DetSim'' branch contains data resulting from the
```

```
  // detector simulation. In the case of real data, it is empty
  EDMDataBucket&  detsim = evt.DetSim();

  // The ''Raw'' branch contains the raw (ie. uncalibrated)
  // detector hit information. This should be essentially the
  // data straight from the DAQ
  EDMDataBucket& raw = evt.Raw();

  // The ''Cal'' branch contains the calibrated detector hit data
  EDMDataBucket& cal = evt.Cal();

  // The ''Reco'' branch contains the results of reconstruction
  // algorithms
  EDMDataBucket& reco = evt.Reco();

  // The MIPPSummary branch is the ''official'' ntuple-style
  // summary of the reconstructed event
  MIPPEventSummary& summary = evt.Summary();

  // The following code shows how to extract a list of RICH digits
  // from the event and print them all
  std::vector<const RICHDigit*> richHits;
  raw.Get(''./rich'',richHits);
  std::vector<const RICHDigit*>::iterator itr(richHits.begin());
  std::vector<const RICHDigit*>::iterator itrEnd(richHits.end());
  for (; itr!=itrEnd; ++itr) {
    std::cout << (*itr)->GetPM();
  }
}
```

Let's take a look at what we have here. There are three classes in play, EDMEventHeader, EDMDataBucket, and MIPPEventSummary. The header and summary are pretty straight forward classes and basically hold several important numbers associated with the event. The EDMDataBucket class (mc, detsim, raw, cal, and reco) is a little more subtle as is it capable of delivering *any* type of object from the event. The basic syntax for extracting data from the event is shown by the "Get" call. Data objects are held in the event in directories (actually ROOT TFolders). So the "Get" call loads the RICH pmt hits from the directory "./rich" into the (STL) vector richHits. Notice the "const" in the vector declaration: users are not allowed to modify

data which is already associated with an event.

2.1.1 Where is the data? edm_dump

The event data structure is very flexible, and like with any hierarchical directory structure it can be hard to keep track where any particular piece of data may be with in an event. To help with this a utility `edm_dump` is provided to print as text the complete contents of an event in a file. Usage looks like:

    edm_dump myfile.root

`edm_dump` takes a few options:

- `-r [n]`: *Dump run number [n]. Default is 1st run in the file*

- `-e [n]`: *Dump event number [n]. Default is 1st event in file*

So

    edm_dump myfile.root

would dump the first event in the file

    edm_dump -r 100 myfile.root

would dump the first event in run 100 in the file and

    edm_dump -r 100 -e 25 myfile.root

would dump event 25 in run 100.

2.1.2 Handling Errors: `EDMException`

Occasionally, a "`Get`" method will fail to load the requested data from an event. This may be relatively benign, for example an attempt to extract MC hits from a real data event. Or it might be fatal, for example a track fitter which cannot find any tracks to fit. The event data model leaves the decision about the severity of a failed "`Get`" to the user by using `C++` exceptions.

Exceptions in `C++` are flags raised by the software. If no action is taken, they will cause the program to abort and dump core. For fatal errors, this is exactly the desired behavior: the program does not silently continue and produce garbage, and it leaves a core file that the user can use to debug the the problem. However, the user's code has the option of checking for these flags, handle them, and allow the program to continue. In the case of the Event-DataModel package, the exceptions are defined by the class `EDMException`.

Again, some sample code will probably make all this more clear:

```
#include "EventDataModel/EDMException.h"
#include "EventDataModel/EDMEventHandle.h"
#include "EventDataModel/EDMDataBucket.h"

void MyClass::MyMethod(const EDMEventHandle& evt)
{
  try {
    // Get a list of DC digits from the "raw" branch in the file
    // The "Get" operation may produce an exception if there are no
    // DC digits in ./dc1
    EDMDataBucket&           raw    = evt.Raw();
    std::vector<const DCDigit*> dclist = raw.Get("./dc1", dclist)
    // Do something useful with the digits...
  }
  catch (EDMException e) {
    // If any EDMExceptions were raised, deal with them here. It
    // probably means that there is no DCDigit data in ./dc1
    // Failure to catch this exception would result in a core dump
    std::cerr << "Failed to load DCDigits!" << std::endl;
    return;
  }
}
```

This show the basic syntax of the `try...catch` syntax in `C++` . The "try" executes a bunch of code, with the knowledge that an exception could occur at any step along the way. If an exception is raised, execution skips to the "catch" block. In this case, the catch is pretty lazy and just reports the error before it ignores it. It could however try to do something more clever to recover.

Chapter 3

Analysis and Reconstruction: anamipp

Reconstruction and analysis of MIPP root data files is performed using a program called "`anamipp`" which is created in the `JobCModule` package. This chapter outlines how users can build and run their own jobs. There are several examples of reconstruction programs one can follow in the repository. The `RICHReco`, `BCReco`, `TPCReco` packages come to mind. The simplest place to start, however, is with the `Demo` package which does no reconstruction, but outlines the basics of getting a module hooked up.

3.1 Getting started with the `Demo` package

Offline jobs are defined as a series of modules (specifically classes inheriting from the class `JobCModule`). Each module performs some piece of analysis or reconstruction, storing its results into the event. To remain pluggable, modules must be kept independent of each other, which is why they are allowed to pass data to the event and not directly to each other.

I could explain how to plug into `anamipp` from either the bottom up (modules to job) or from the top down (jobs down to modules). I'm going to go from the top down.

This chapter is something of a tutorial so pull up a terminal window and get ready to type-along...

3.1.1   Running `anamipp`

The first thing to do is to set up your SRT test release and then check the
`Demo` package out into your release:

```
% srt_setup -a
% addpkg -h Demo
```

Once that's done, lets change into the `Demo` package and make a link to a
root file to play with:

```
% cd Demo
% ln -s /path/to/root/file/file.root infile.root
```

That will serve as our input file for the rest of this chapter.
   Now, let's just run the job:

```
% which anamipp
/usr/local/mipp/releases/development/bin/Linux2.4-GCC/anamipp
% anamipp -n 5 -x demojob.xml infile.root
JCXML: Linked libDemo.so
** Start job 'DemoJob' Thu Dec 23 09:55:57 2004 **
Files in list:
>[0] infile.root (i=0)
DemoModule::DemoModule()
DemoModule::Update()
 fInt    = 907
 fFloat  = 0.907
 fDouble = 0.907709
 fString = E907
Set watch for DemoModuledefault
DemoModule::Reco() fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
DemoModule::Reco() fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
DemoModule::Reco() fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
DemoModule::Reco() fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
DemoModule::Reco() fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
** Thu Dec 23 09:55:57 2004 End job DemoJob **
+-DemoJob sequence (5/0/5 events 0u/0s seconds)
 +DemoModule/default/            +reco/00005/00000/00005 0.00e+00/0.00e+00
                                 +ana /00005/00000/00005 0.00e+00/0.00e+00
DemoModule::~DemoModule()
```

The "which" command just assures you that you've got the anamipp job in your path. The next command is where the money is. This runs five events (`-n 5`) through the job detailed in the `demojob.xml` file. Working from the top down, the next step is to look at what goes into a job XML file.

3.1.2   Job description XML documents

Take a look at the `demojob.xml` file:

```
% less demojob.xml
```

This file does basically three things. First, it tells `anamipp` to link in the `Demo` package. The `anamipp` contains all the boilerplate for file I/O and the like, but it contains no reconstruction packages. The `<link>` tag pulls in the reconstruction code.

The second thing that happens is that additional XML files are pulled in. In this case the file `DemoModule.xml` is read to get the configuration parameters to use in the `DemoModule` module. We'll talk about configuration documents a little later on.

The third thing is to actually build the job. A job is a series of "nodes" which are associated with a named job module. Modules typically need some tunable parameters to perform their work. These are stored in configurations which share the module's name, but may come in several different versions. To select the version you want in your job, use the "config" parameter. All modules must have a "default" version defined, so if you don't know where to start, start there.

Modules can perform either a piece of event reconstruction ("reco") or analysis ("ana") on an event, the distinction being that reconstruction adds information to an event (requiring read/write access) while analysis only looks at data in the event (requiring only read access). This might be useful, for example, in a tracking module. The reconstruction could be performed in the "reco" stage and analysis of performance, say compared to Monte Carlo truth, in the "ana" stage. While running on real data or in production one could turn the "ana" stage off and save some CPU time.

Nodes are allowed to make a pass/fail decisions about events. Events which are failed are not processed further. The user has some choices to control how exactly the pass/fail decision is used which are controlled by the "filter" setting. The choices are "on" in which case the pass/fail decisions are enforced, "off" in which case the pass/fail decisions are ignored, and

"reverse" in which case the pass/fail decisions are enforced using "pass" to mean "fail" and "fail" to mean "pass". This would be useful, for example, to output all the events that fail a certain module's selection criteria.

3.1.3   Output from anamipp

The `anamipp` job prints some information for every run. Some of it is self-explanatory. However, the following probably needs some explaining:

```
+-DemoJob sequence (5/0/5 events 0.02u/0s seconds)
  DemoModule/default/          +reco/00005/00000/00005 2.00e-02/0.00e+00
                               +ana /00005/00000/00005 0.00e+00/0.00e+00
```

The first line says that the job "DemoJob" was run. Five events passed the job, 0 events failed, and a total of 5 events were processed (5/0/5). The job used 0.02 seconds of user time on the CPU and 0 seconds of system time on the CPU.

The next lines report what happened in the individual nodes. In this case, the reco and ana methods were called and passed 5 events, failed 0 events, and saw a total of 5 events. The reco method consumed 2e-2 seconds of user CPU time and used 0 seconds of system CPU time.

That's the top layer. Now let's talk about the nitty-gritty of modules.

3.2   Job Modules

To plug your own code into `anamipp` you'll need to write your own job module. The `DemoModule.h` and `DemoModule.cxx` files implement a very simple module which just prints its configuration for each event. These files are pretty well commented and I've tried to put only the essential features in. There are a few things you are required to do to get plugged in. They are labeled in the files:

```
% grep Required DemoModule.h DemoModule.cxx
DemoModule.h:class DemoModule : public JobCModule // Required!
DemoModule.h:  DemoModule(const char* version); // Required!
DemoModule.h:  void Update(const CfgConfig& c); // Required!
DemoModule.cxx:MODULE_DECL(DemoModule); //Required!
DemoModule.cxx:  JobCModule("DemoModule") // Required!
DemoModule.cxx:  this->SetCfgVersion(version); // Required!
```

3.2.1 DemoModule.h

- `class DemoModule : public JobCModule`

  A module is required to inherit from the class `JobCModule` which defines the generic interface to job modules used by `anamipp`.

- `DemoModule(const char* version)` Modules are configurable objects so they are required to implement a constructor which takes a version tag labeling the configuration that should be used to set the module up. Users should use this and only this constructor in their modules.

- `void Update(const CfgConfig& c)` Again, since modules are configurable items, they must implement the `Update` method (inherited indirectly from the `CfgObserver` class) which is called when the configuration used by the module changes requiring the module to update its data.

3.2.2 DemoModule.cxx

- `MODULE_DECL(DemoModule);`

  This macro declares your module to `anamipp` so that it can be linked into a job using your chosen name for it.

- `JobCModule("DemoModule")`

  This appears in the constructor. You are required to call the JobCModule constructor using your chosen name for the module.

- `this->SetCfgVersion(version)`

  This also appears in the constructor. You are required to inform the base classes which configuration version you are using.

That's about it for required code. You have the option of implementing the `Reco` and `Ana` methods which will actually do work. These are given access to the data using the `EDMEventHandle` class which is discussed in the earlier chapter on the data model. For other examples:

```
% grep evt $SRT_PUBLIC_CONTEXT/RICHReco/RICHCircFit.cxx
% grep evt $SRT_PUBLIC_CONTEXT/RICHReco/RICHRadFit.cxx
```

The `DemoModule::Reco` method provides some sample code to show how to store objects (`TLine`'s as an example) on the event.

Since we talked about how to store objects, let's prove that the `DemoModule` is in fact storing its `TLine`'s on the event:

```
% anamipp -n 1 -o outfile.root -x demojob.xml infile.root
JCXML: Linked libDemo.so
** Start job 'DemoJob' Thu Dec 23 11:28:06 2004 **
Files in list:
>[0] infile.root (i=0)
DemoModule::DemoModule()
DemoModule::Update()
 fInt    = 907
 fFloat  = 0.907
 fDouble = 0.907709
 fString = E907
Set watch for DemoModuledefault
DemoModule::Reco()
fInt=907, fFloat=0.907, fDouble=0.907709, fString=E907
** Thu Dec 23 11:28:06 2004 End job DemoJob **
+-DemoJob sequence (1/0/1 events 0u/0s seconds)
  DemoModule/default/            +reco/00001/00000/00001 0.00e+00/0.00e+00
                                 +ana /00001/00000/00001 0.00e+00/0.00e+00
DemoModule::~DemoModule()
% edm_dump outfile.root
Dump of run = 10603 event = 0 file = outfile.root
evt.Header()
|================================================================
evt.DetSim()
|-* hits
|================================================================
evt.MC()
|-* kine
|================================================================
evt.Raw()
|-* trig
|-* bckov
|-* dc1
|-* dc2
  |-* DCDigits[98]
|-* dc3
```

```
   |-* DCDigits[107]
|-* dc4
   |-* DCDigits[7]
|-* bc1
   |-* DCDigits[116]
|-* bc2
   |-* DCDigits[4]
|-* bc3
   |-* DCDigits[5]
|-* ecal
   |-* ADCDigits[640]
|-* hcal
   |-* ADCDigits[8]
|-* mwpc1
|-* mwpc2
|-* rich
   |-* RICHDigits[28]
|-* t0
|-* tof
|-* tpc
   |-* TPCStickInfos[79]
   |-* TPCDigits[7114]
|-* ckov
|================================================================
evt.Cal()
|================================================================
evt.Reco()
|-* demo
   |-* demomodule
      |-* TLines[10]
|================================================================
evt.User()
|================================================================
evt.Summary()
|================================================================
```

This shows the use of the very useful edm_dump (edm_dump --help for more information)program which dumps the event structure stored in a file. In this case, you'll notice that 10 TLine's have appeared in the Reco tree in the /demo/demomodule folder.

3.2.3 Other methods

Job modules can also implement methods to receive notice at the beginning and end of runs and sub-runs, at the beginning and end of files and to print status reports or reset themselves. These are all outlined in:

```
% less $SRT_PUBLIC_CONTEXT/JobCModule.h
```

3.2.4 Configuration

As mentioned previously, modules are configurable objects. That is, they typically use parameters to make decisions, fits, and otherwise do their job. These configuration parameters should not be hard coded but rather stored using the mechanism setup by the `Config` package. This will allow you, for example, to tweak the module's configuration parameters and rerun your fit and look at the result from the event display – a fantastically useful tool for tuning reconstruction programs.

You are required to supply a default configuration for your module. The file "DemoModule.xml" does this for the `DemoModule`. Its pretty well commented, so I'll let it explain itself. This file works in conjunction with the code in `DemoModule::Update` to get the parameters defined in the configuration loaded into the module.

Chapter 4

Monte Carlo

MIPP Monte Carlo generation is a 2-step process. First one has to generate a StdHep file[1] using an external package such as FLUKA or DPMJET. Then `e907mc` executable (compiled in e907mc package), can be run on the StdHep file, tracking particles using GEANT 3.21. The output of e907mc is a ROOT file which contains hits (local energy depositions) and particle stack. One can specify that any number of detectors be digitized (mock raw data generated) at the same time.

4.1 Event Generation

MIPP software provides for two ways of generating events: using FLUKA or DPMJET.

4.1.1 FLUKA

TODO

4.1.2 DPMJET

TODO

---

[1]StdHep file must be binary, i.e. written through libFmcfio. Use StdHep 5.05.05 or later, available from `http://cepa.fnal.gov/psm/stdhep/` or `ftp://ftp.fnal.gov/products/stdhep/`. Text StdHep files written through CLHEP cannot be read by e907mc

4.2 `e907mc`

Monte Carlo is based on GEANT 3.21, but the program itself contains no hard-coded geometry or configuration numbers. `e907mc` is configured through a combination of RCP files and XML files, so virtually no geometry constants are compiled into the application. This section describes how `e907mc` can be configured and run.

4.2.1 Execution details

Execution path is something along these lines

1. Command-line options are parsed;

2. e907mc.xml file is parsed;

3. Soft links are created to rcp(e) files, including `target_rcp` link pointing to the right target geometry file;

4. `hanuman_rcpe` file is created from templated `hanuman_generate.rcpe` or `hanuman_interactive.rcpe` depending on whether interactive or batch mode were selected;

5. Bfield is configured based on the run number (default is field on);

6. ROOT geometry is loaded for selected target;

7. ROOT file is created using the run number (can be overridden from command line);

8. Execution is passed to GEANT side through `e907mc_()` subroutine (defined in hanuman.F)

9. `mci_writeevt_()` function is called in `guout.F` (defined in `dd_geant` package). This function makes calls to `E907MCInterface` modules which extract the particle stack, hit stacks for various detectors, and perform digitization. All objects are written to ROOT event stream (EventDataModel);

10. `TheEnd()` function defined in `e907mc.cc` is called at the very end. ROOT file gets closed and soft links are cleaned up.

4.2.2 How to Run It

In the simplest case, one would start Monte Carlo generation with
`e907mc -s <input.stdhep>`
The number of events generated is determined by `TRIG` field in `hanuman.rcp`
file or the maximum number of events read from StdHep file. The following
options are available to fine tune running:

-n [`--nevt`] Followed by number of triggers to generate;

-s [`--stdhep`] Followed by StdHep file for input kinematics;

-r [`--rzout`] Followed by the name of the output GEANT zebra file;

-g [`--histo`] Followed by the name of PAW histogram file;

-o [`--output`] Followed by the name of ROOT output file. By default
run/subrun number is requested from the DB, and the output file is
named accordingly;

-p [`--prompt`] Run in interactive mode;

-t [`--target`] Followed by target type name: cryo (default), numi, or
wheel are recognized;

-R [`--run`] Followed by run number based on which Monte Carlo will be
setup.

4.2.3 Configuring Execution

One should assume that out of the box configuration of `e907mc` is reason-
able. However, for specialized studies, one will have to tweak at least some
parameters.

A number of RCP files exist in `e907mc` and `beamline` packages, which
setup geometry and hit definitions on the GEANT side. Some of these pa-
rameters (e.g. hit array definitions) can affect C++ side of the code in a
bad way, so be careful changing parameters. You should also know that
experiment geometry is setup for GEANT and C++ side through different
macros: RCP files control GEANT geometry, and ROOT macros in `Geometry`
package control ROOT geometry (the latter are generated from the former).
Therefore, if you want to move a chamber, it will not be sufficient to move

it in RCP file. You will have to propagate the changes to the corresponding `Geometry` macro.

Controlling Digitization   As of this writing, by default `e907mc` only stores GEANT hits and stores RICH digits (because that Fortran code depends on GEANT common blocks. The advantage of this method lies in the fact that GEANT tracking takes a significant amount of time, hence digitization is much easier tuned with hits already written to disk. All detectors are digitized in `JobCModule`'s with names like `ChamDigitizer`, etc. In order to include a digitizer into `e907mc`, it must be listed in `MCIDoDigi.xml` file of `E907MCInterface` package. By default, only RICH is listed. Other existing digitizers are `Cham`, `TPC`.

4.2.4   Reconstructing Monte Carlo

Objects contained in data stream after digitization should resemble raw data to the best extent possible, hence "usual" reconstruction algorithm should work. Currently, the only exception is lack of simulation of the beamline, so trigger information and beam chamber response is not available. If you use default MC file (with RICH digits only), then digitization and reconstruction can be done in the same job. A good example of job description is `mcRecoJob.xml`, which includes

- `DigiSequence` to digitize chambers, and the TPC,

- `TrkRBase` sequence to reconstruct chamber data,

- `TPCReco` sequence to reconstruct TPC data,

- `SPFit` modules to connect TPC and chamber data into global tracking,

- `FillEventSummary` module to build DST.

Chapter 5

The MIPP Event Display: evd

The MIPP event display (executable "evd") is a very powerful tool for visualizing events in the MIPP spectrometer. In the event display, ne can draw both the raw data and reconstructed hits, tracks and vertices, allowing for data quality monitoring as well as a means to easily (visually) verify the results of any event reconstruction algorithm.

5.1   Starting the Event Display

The name of the executable for the MIPP event display is "evd". The command line arguments for evd are the same as anamipp's. The event display is configured via XML, the arguments of which are in evd. xml. If one wishes to use a different startup configuration for the event display, edit the evd. xml file and start evd with the command:

```
> evd -x evd.xml [input file]
```

5.2   Event Display Windows

The event display offers a variety of windows to display MC hits, raw data (both MC and real) and reconstructed hits, tracks and vertices. Windows may be opened up by selecting them from the "Window" menu. Often one wishes to see only data in a single detector, so there are display windows for nearly every detector in the MIPP experiment. The display in some windows are configurable via XML parameters. To change the display settings while

running the event display, select the desired window from the "Edit" menu. Below, we describe a few windows that have proven to be most useful to the general user.

5.2.1 Full Display

The "Full Display" window displays the front, side and top view of the full MIPP spectrometer in wire frames. In addition there is a 3D view of the spectrometer that can be rotated and zoomed. In this display window, all raw digits and reconstructed tracks in the MIPP event tree are drawn.

5.2.2 Tracking Display

The "Tracking Display" is another display for viewing hits in the full MIPP spectrometer. Here only reconstructed hits, tracks and vertices are shown. Because of its cleaner presentation of the TPC and downstream wire chamber volumes, this display will be used in handscan studies of reconstructed events. It is relatively easy for the user to determine whether or not a reconstructed TPC track is matched with the appropriate downstream hits, or vice-versa.

5.2.3 TPC Display

The "TPC Display" shows MC tracks, raw TPC digits, reconstructed TPC hits, and reconstructed tracks and vertices in the TPC volume only. Because the target is located so close to the TPC volume, this is a good display to use to verify the vertex finding and fitting algorithms, as well as the TPC-specific reconstruction algorithm. By default, the display of MC tracks and raw hits is disabled; one can display these by changing the XML parameters of the TPC display options.

5.2.4 MC Information

This window contains a list of all MC tracks in the event, with a fairly complete description of each track (momentum, particle type, mother particle, etc.). One can select which tracks are drawn by clicking on the checkbox next to each particle. If unchecked, the track will not be drawn in any of the above displays. If checked, the track will only be drawn in some displays if the corresponding window's display options allow MC hits to be drawn.

5.3 Running a Job in the Event Display

One of the most powerful uses of the MIPP event display is the ability to run
and configure a reconstruction or analysis job in real time, event-by-event,
in the event display. To do so, select "Open XML" in the "Job" menu,
and select the desired job description XML file from the file dialog. If the
event display window is designed to do so, the reconstructed objects (eg, hits,
tracks, vertices) will be drawn in the window.

To change the configuration of any of the modules used in the job, select
the desired module from the "Edit Config" option in the "Job" menu. A
dialog will appear with all of the XML parameters of that module; after
making the desired changes to the values of the parameters, click on "Apply"
and the job sequence will run again and the event will be redrawn.

Chapter 6

Data Analysis with the DST

A condensed version of the event reconstruction is available via the Data Summary Table (DST). The DST is essentially a Root `TTree`; currently, two branches of the tree are filled, one using results obtained from the simple TPC reconstruction in the `TPCRecoJP` module (helix fits to tracks), the other using results obtained from the global track fits in the `SPFit` module. One should not, however, that detector PID information is *o*nly available in the `SPFit` branch. Therefore, most analyses should use the "SPReco" branch of the DST TTree.

6.1 Format of the DST - Round 2

A schematic of the format of the DST used in Round 2 (completed early 2006) of the reconstruction is shown in Fig. 6.1. Each event in the DST is described by the `MIPPEventSummary` class. This class holds general information such as run and event number, raw trigger information, some raw digit information, and lists of pointers to `MIPPTrackSummary`'s and `MIPPVertexSummary`'s. These objects contain information for each reconstructed track or vertex as described in the subsections below.

6.1.1 The `MIPPEventSummary` Class

Some general information about the event in question may be accessed through the public members of the `MIPPEventSummary` class. The list and description of these variables are listed in Table 6.1.
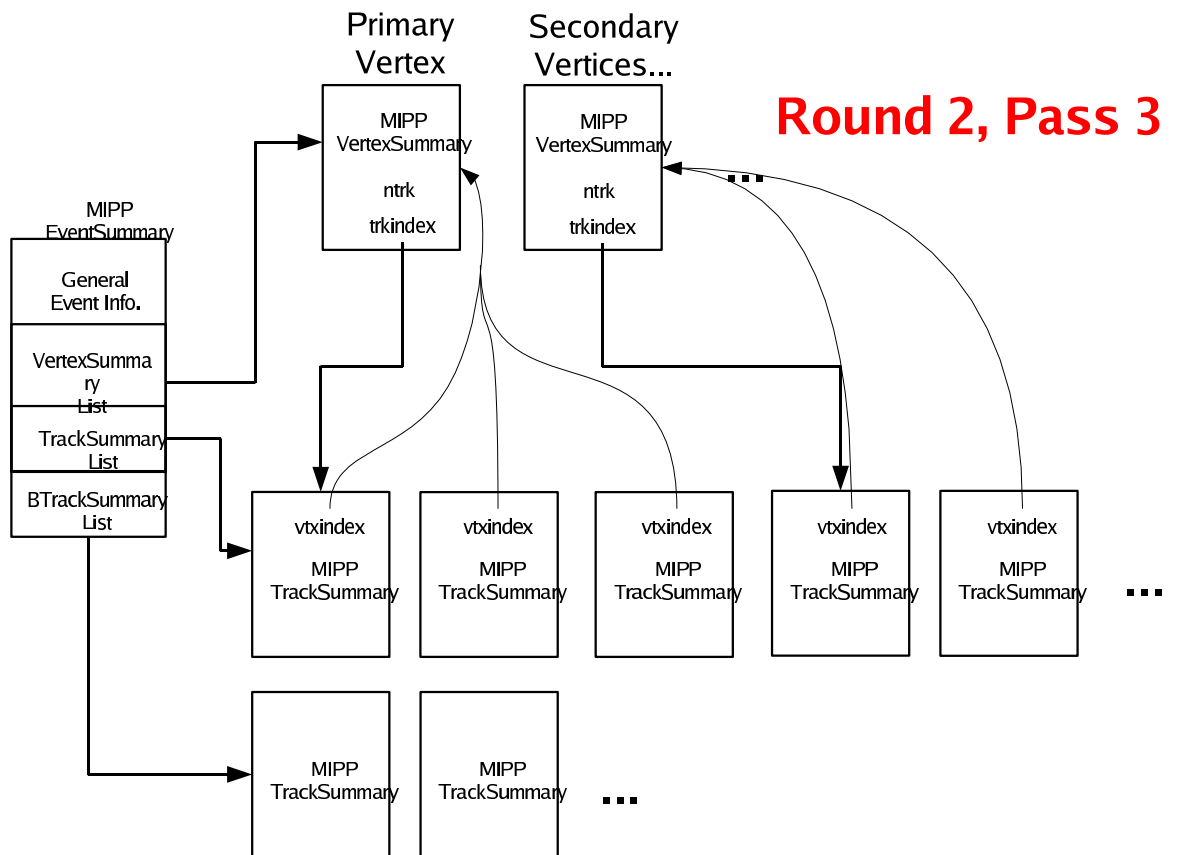
Figure 6.1: Schematic of the DST format used in Round 2.

6.1.2 The MIPPTrackSummary Class

A summary of a reconstructed track in the event is held in the `MIPPTrackSummary` class. This class is currently used to describe both incoming beam tracks as well as tracks emanating from or downstream of the target. The momentum of the incoming beam track is, at this point, simply taken from the database value of the average momentum for the run. The particle ID of the beam track, however, comes from the results of the `BCkovReco` module. The particle ID detector information for downstream tracks is gathered from `TOFReco`, `CkovReco`, `RICHReco`, and `TPCRdEdx`.

The list and description of the variables in this class are listed in Tables 6.2 and 6.3.

6.1.3 The MIPPVertexSummary Class

A summary of a reconstructed vertex in the event is held in the `MIPPVertexSummary` class. By convention, the first `MIPPVertexSummary` object in the array held by the mother `MIPPEventSummary` object is the primary vertex. This class is fairly straightforward, and it contains the index of the first track associated with it and the number of tracks associated with it, so that one can easily loop over all tracks associated to a given vertex.

The list and description of the variables in this class are listed in Table 6.4.

6.2 Format of the DST - Round 3

A schematic of the format of the DST for round 3 (to be completed mid 2006) of the reconstruction is shown in Fig. 6.2. Here, the information from a particle ID for a matched track is stored in a separate object (class). As before, general event information is stored in the `MIPPEventSummary` class, generic track information is stored in the `MIPPTrackSummary` class and vertex information is stored in the `MIPPVertexSummary` class.

Also new to this version of the DST is a separate TTree holding spill information. Basically this TTree holds the information of the scalar readouts after each spill in the MIPP experiment necessary for cross-section measurements. The information is held in the `MIPPSpillSummary` class describe below.

Class descriptions are listed in Tables 6.5-6.13.

Round 3, Pass 3
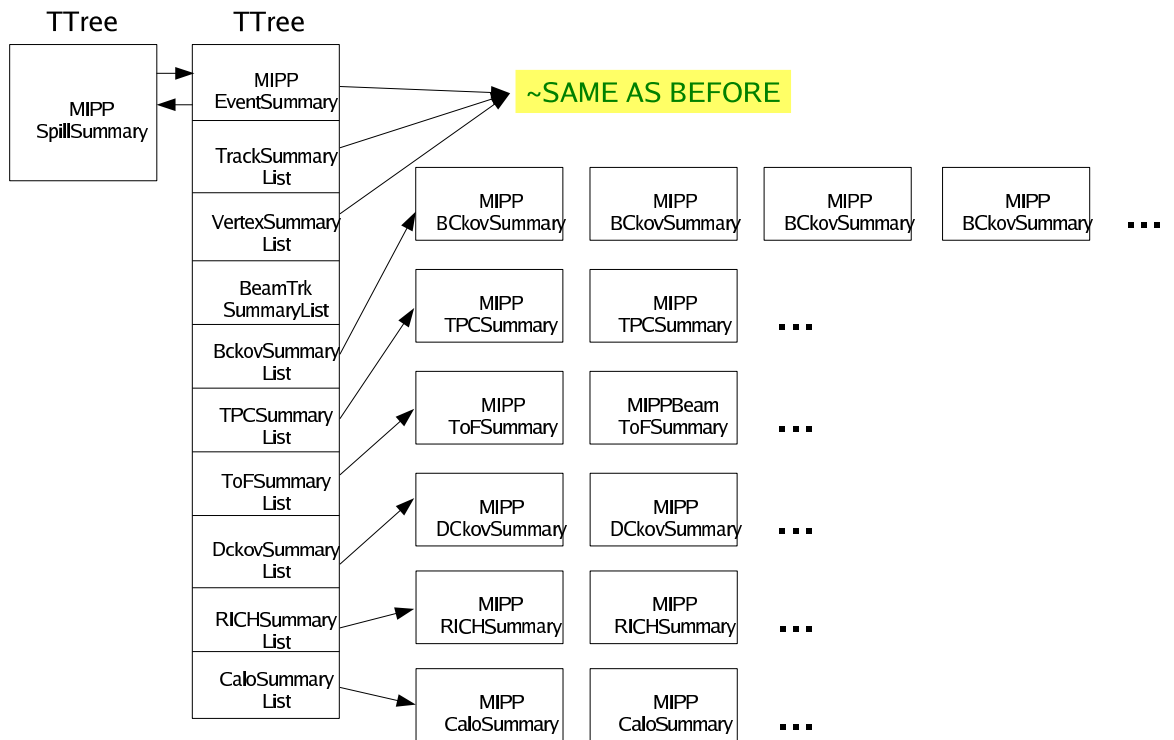
Figure 6.2: Schematic of the DST format to be used in Round 3.

6.3   Getting Started With the DST

A simple script ("mkDSTmacro") is installed in the MIPP offline software release that dumps a template macro to the screen. For completeness, here is the macro:

```
#include <iostream.h>

void DSTTemplate(char fname[256], int numevents=0)
{

  if (fname == "") {
    std::cout << "Usage: DSTTemplate.C([DST File Name],[nevents])"
              << std::endl;
    return;
  };

  char mytxt[256];
  char outfname[256];

  gROOT->Reset();
  gStyle->SetOptTitle(1);
  gSystem->Load("libMIPPEventSummary");

  MIPPEventSummary*  evt = new MIPPEventSummary();
  MIPPVertexSummary* vtx;
  MIPPTrackSummary*  trk;

  TFile* DSTFile = new TFile(fname);
  TTree* DSTTree = (TTree*)DSTFile->Get("FillEventSummary/fNtTree");

  //***** if you want results from SPReco, comment out the
  //***** next line and uncomment the line below it

  TBranch* DSTBr = DSTTree->GetBranch("fTPCReco");
  //  TBranch* DSTBr = DSTTree->GetBranch("fSPReco");

  DSTBr->SetAddress(&evt);
```

```
DSTTree->SetBranchAddress("fDST",&evt);

int nevt = (int)DSTTree->GetEntries();

if (numevents <= 0 || numevents > nevt) numevents = nevt;

for (int ievt = 0; ievt<numevents; ++ievt) {
  evt->Clear();
  DSTTree->GetEntry(ievt);

  if (ievt == 0) std::cout << "Processing run " << evt->run << std::endl;

  int ntrk = evt->NTrk();
  int nvtx = evt->NVtx();

  std::cout << ntrk << " tracks in event " << evt->evtnum << std::endl;
  std::cout << nvtx << " vertices in event " << evt->evtnum << std::endl;

  // here's how to loop through all vertices in an event:
  for (int ivtx=0; ivtx < nvtx; ++ivtx) {
    vtx = evt->GetVtx(ivtx);
    std::cout << "\t Vertex " << ivtx << " at (" << vtx->x[0] << ","
              << vtx->x[1] << "," << vtx->x[2] << ") has " << vtx->ntrk
              << " tracks." << std::endl;
    // now do something with the vertex...
  }

  // here's how to loop through all tracks in an event:
  for (int itrk=0; itrk < ntrk; ++itrk) {
    trk = evt->GetTrk(itrk);
    std::cout << "\t Track " << itrk << " belongs to vertex "
              << trk->vtxindex << std::endl;
    // now do something with the track...
  }

  // here's how to loop through all tracks belonging to
  // each vertex in an event:
```

```
    for (int ivtx=0; ivtx < nvtx; ++ivtx) {
      vtx = evt->GetVtx(ivtx);
      int vntrk = vtx->ntrk;
      int itrk = vtx->trkindex;
      for (int j=0; j<vntrk; ++j) {
        trk = evt->GetTrk(itrk+j);
        std::cout << "\t\t Track " << (itrk+j) << " has momentum = "
                  << trk->ptot << " GeV" << std::endl;
      }

    }

  }

}
//*******************************************************************
```

To create a macro of your own from this script, simple execute:

```
> mkDSTmacro > myDSTmacro.C
```

and edit the macro to fit your needs. The macro can then be used in an interactive Root session. If one desires faster processing, it should be easy modify the script to be an executable, although at the time of this writing, this has yet to be done.

Summary of `MIPPEventSummary` Class - Round 2			
Var. name	Var. type	Default value	Description
run	I	0	Run number
evtnum	I	-1	Event number
rawtrig	I	-1	Raw trigger word
pstrig	I	-1	Prescale trigger word
sciadc	I	0	Scint. Trigger ADC value
scitdc	I	0	Scint. Trigger TDC value
nwbc	I	0	Num. hit BC wires
npbc[3]	I	0	Num. hit BC planes, for each BC
nwdc1	I	0	Num. hit DC1 wires
npdc1	I	0	Num. hit dc1 planes
ntofbars	I	0	Num. hit tof bars
tofbar[16]	I	-1	Array of hit tof bar numbers
nrr	I	0	Num. RICH rings (rr)
nckovm	I	0	Num. DCkov mirrors above threshold
t0	F	-1.e5	start time (taken at T01)
rrpos[2][16]	F	-1.e5	center (x,y) position of each RICH ring
rrrad[16]	F	-1.	radius of each RICH ring
avgb[2]	F	0.	avg. B field (0 = JGG, 1 = Rosie)
tgta	F	0.	target atomic number
GetTrk(i)	T*	0	method for getting ptr to track `i`
GetBTrk(i)	T*	0	method for getting ptr to beam track `i`
GetVtx(i)	V*	0	method for getting ptr to vertex `i`

Table 6.1: Description of the `MIPPEventSummary` class. Var. type I == integer, F == float, T = MIPPTrackSummary, V = MIPPVertexSummary.

Summary of `MIPPTrackSummary` **Class** (Track Information) - Round 2			
Var. name	**Var. type**	**Default value**	**Description**
vtxindex	I	-1	index of associated vertex, outgoing
invtxindex	I	-1	index of associated vertex, incoming
pid	P	0	paricle id
pid_def	D	0	PID detector id
nwwc	I	0	Num. of WC wires in track
npwc	I	0	Num. of WC planes in track
q	I	0	charge of particle
x[3]	F	-1.e5	position vector at tgt
dx[3]	F	-1.e5	pos. uncertainty at tgt
p[3]	F	-1.e5	momentum vector at tgt
dp[3]	F	-1.e5	mom. uncertainty at tgt
ptot	F	-1.e5	total momentum at tgt
pt	F	-1.e5	transverse mom. at tgt
dpid[4][5]	F	-1.e5	Log-likelihood PID matrix row=PID::Detector_t, col=PID::PID_t
gof	F	0.	goodness-of-fit

Table 6.2: Description of the `MIPPEventSummary` class. Var. type I = integer, F = float, P = `RBPID::PID_t`, D = `RBPID::Detector_t`.

Summary of `MIPPTrackSummary` Class (PID Detector Information) - Round 2			
Var. name	**Var. type**	**Default value**	**Description**
TPC			
ntpchits	I	0	Num. of TPC hits in track
ndedx	I	0	Num. of TPC hits used in $\langle$ dE/dx $\rangle$
tpcdedx	F	0.	$\langle$ dE/dx $\rangle$
ToF : (0=ToF Bar, 1=ToF Calib.Bar), (top=0, bottom=1)			
tofbar[2]	I	0	associated ToF bar number
toftdc[2][2]	I	0	ToF bar TDC
tofadc[2][2]	I	0	ToF bar ADC
tofx(y)[2]	F	-1.e5	projected x(y)-pos. at ToF
tofds[2]	F	0.	pathlength from target center to ToF
DCkov : 3x3 mirror grid around central mirror			
ckovm	I	-1	DCkov central mirror id
ckovntdc[3][3]	I	0	Num. TDCs in mirrors
ckovtdc[3][3][10]	F	0	TDC values of 3x3 mirror grid
ckovadc[3][3]	F	0	ADC values of 3x3 mirror grid
ckovx(y)	F	0.	$x(y)$-pos. at upstream end of DCkov
ckovxp(yp)	F	0.	$dx(y)/dz$ at upstream end of DCkov
ckovds	F	0.	path length of track inside DCkov
RICH			
rmirr	I	-1	projected RICH mirror
rrad	F	-1.e5	associated ring radius
drrad	F	-1.e5	uncertainty of ring radius
richx(y)	F	-1.e5	ring center, (x,y)
rtrkx(y)	F	-1.e5	position at RICH

Table 6.3: Description of the `MIPPEventSummary` class. Var. type I = integer, F == float

Summary of MIPPVertexSummary Class - Round 2			
Var. name	**Var. type**	**Default value**	**Description**
ntrk	I	0	Num. tracks in vertex
trkindex	I	-1	index of 1st track in vertex
intrkindex	I	-1	index of incoming track
vtxstatus	I	0	vertex fit status
vtxtype	I	0	vertex type (1==Primary, 2==Secondary, etc.)
x[3]	F	-1.e5	vertex position vector
dx[3]	F	-1.e5	vertex pos. uncertainties
ptot	F	1.e5	summed momentum of charged tracks
pt	F	1.e5	summed transverse mom. of charged tracks
pz	F	1.e5	vertex total p_z
gof	F	0.	goodness-of-fit

Table 6.4: Description of the `MIPPEventSummary` class. Var. type I = integer, F == float.

Summary of MIPPSpillSummary Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
run	I	0	Run number
spill	I	0	Spill number
nevt	I	0	Number of events in spill
rawTrigInput[40]	I	0	raw trigger scalar count
gatedTrigBit[20]	I	0	gated trigger bit scalar count
rawTrigBit[20]	I	0	raw trigger bit scalar count
psTrigBit[20]	I	0	prescaled trigger bit scalar count
psAndGatedTrigBit[20]	I	0	prescaled and gated trigger bit scalar count

Table 6.5: Description of the `MIPPSpillSummary` class used in Round 3. Var. type I == integer.

Summary of `MIPPEventSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
run	I	0	Run number
spill	I	-1	Spill number
evtnum	I	-1	Event number
rawtrig	I	-1	Raw trigger word
pstrig	I	-1	Prescale trigger word
bckov1adc[2]	I	-1	Beam Ckov 1 ADCs
bckov2adc[2]	I	-1	Beam Ckov 2 ADCs
bckov1tdc[2]	I	-1	Beam Ckov 1 TDCs
bckov2tdc[2]	I	-1	Beam Ckov 2 TDCs
sciadc	I	0	Scint. Trigger ADCs
scitdc	I	0	Scint. Trigger TDCs
nwbc	I	0	Num. hit BC wires
npbc[3]	I	0	Num. hit BC planes, for each BC
nwdc1	I	0	Num. hit DC1 wires
npdc1	I	0	Num. hit dc1 planes
ntofhits	I	0	Num. tof hits
ntofbars	I	0	Num. hit tof bars
nckovm	I	0	Num. DCkov mirrors above threshold
t0	F	-1.e5	start time (taken at T01)
avgb[2]	F	0.	avg. B field (0 = JGG, 1 = Rosie)
tgta	F	0.	target atomic number
GetTrk(i)	T*	0	method for getting ptr to track `i`
GetBTrk(i)	B*	0	method for getting ptr to beam track `i`
GetVtx(i)	V*	0	method for getting ptr to vertex `i`
GetTPC(i)	TPC*	0	method for getting ptr to TPC trk info `i`
GetToF(i)	ToF*	0	method for getting ptr to ToF trk info `i`
GetDCkov(i)	DCkov*	0	method for getting ptr to DCkov trk info `i`
GetRICH(i)	RICH*	0	method for getting ptr to RICH trk info `i`
GetCalo(i)	Calo*	0	method for getting ptr to Calo trk info `i`

Table 6.6: Description of the `MIPPEventSummary` class. Var. type I == integer, F == float, T = MIPPTrackSummary, B = MIPPBeamTrkSummary, V = MIPPVertexSummary, TPC = MIPPTPCSummary, etc.

Summary of `MIPPTrackSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
vtxindex	I	-1	index of associated vertex, outgoing
invtxindex	I	-1	index of associated vertex, incoming
pid	P	0	paricle id
pid_def	D	0	PID detector id
q	I	0	charge of particle
nwchits[6]	I	0	number of wire hits in track
itpc	I	-1	index of assoc. MIPPTPCSummary
itof	I	-1	index of assoc. MIPPToFSummary
idckov	I	-1	index of assoc. MIPPDCkovSummary
irich	I	-1	index of assoc. MIPPRICHSummary
icalo	I	-1	index of assoc. MIPPCaloSummary
x[3]	F	-1.e5	position vector at tgt/vtx
dx[3]	F	-1.e5	pos. uncertainty at tgt/vtx
p[3]	F	-1.e5	momentum vector at tgt/vtx
dp[3]	F	-1.e5	mom. uncertainty at tgt/vtx
ptot	F	-1.e5	total momentum at tgt/vtx
pt	F	-1.e5	transverse mom. at tgt/vtx
gof	F	0.	goodness-of-fit
utpcx[3]	F	-1.e5	pos. vector at upstream face of TPC
utpcp[3]	F	-1.e5	mom. vector at upstream face of TPC
dtpcx[3]	F	-1.e5	pos. vector at downstream face of TPC
dtpcp[3]	F	-1.e5	mom. vector at downstream face of TPC
urosyx[3]	F	-1.e5	pos. vector at upstream face of ROSY
urosyp[3]	F	-1.e5	mom. vector at upstream face of ROSY
drosyx[3]	F	-1.e5	pos. vector at downstream face of ROSY
drosyp[3]	F	-1.e5	mom. vector at downstream face of ROSY
wcx[6][3]	F	-1.e5	pos. vector at downstream face of WC 1-6
wcp[6][3]	F	-1.e5	mom. vector at downstream face of WC 1-6

Table 6.7: Description of the `MIPPEventSummary` class. Var. type I = integer, F = float, P = `RBPID::PID_t`, D = `RBPID::Detector_t`.

Summary of MIPPVertexSummary Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
ntrk	I	0	Num. tracks in vertex
trkindex	I	-1	index of 1st track in vertex
intrkindex	I	-1	index of incoming track
vtxstatus	I	0	vertex fit status
vtxtype	I	0	vertex type (1==Primary, 2==Secondary, etc.)
x[3]	F	-1.e5	vertex position vector
dx[3]	F	-1.e5	vertex pos. uncertainties
p[3]	F	0.	vector sum of track momenta
dp[3]	F	0.	uncertainties of vector sum of track momenta
ptot	F	1.e5	summed momentum of charged tracks
pt	F	1.e5	summed transverse mom. of charged tracks
gof	F	0.	goodness-of-fit

Table 6.8: Description of the `MIPPEventSummary` class. Var. type I = integer, F = float.

Summary of MIPPTPCSummary Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
itrk	I	0	Track index
nhits	I	0	Num. of TPC hits in track
ndedx	I	0	Num. of TPC hits used in $\langle dE/dx \rangle$
dedx	F	0	$\langle dE/dx \rangle$ of track
LL[5]	F	0	Log-likelihood PID for different particle species

Table 6.9: Description of the `MIPPTPCSummary` class. Var. type I = integer, F = float.

Summary of `MIPPToFSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
itrk	I	0	Track index
bar	I	0	associated ToF bar
cbar	I	0	associated ToF calib. bar
tadc	I	0	ToF bar top ADC
ttdc	I	0	ToF bar top TDC
badc	I	0	ToF bar bottom ADC
btdc	I	0	ToF bar bottom TDC
ceadc	I	0	ToF calib. bar east ADC
cetdc	I	0	ToF calib. bar east TDC
cwadc	I	0	ToF calib. bar west ADC
cwtdc	I	0	ToF calib. bar west TDC
x	F	-1.e5	projected x-pos. of track at ToF bar
y	F	-1.e5	projected y-pos. of track at ToF bar
cx	F	-1.e5	projected x-pos. of track at ToF calib. bar
cy	F	-1.e5	projected y-pos. of track at ToF calib. bar
ds	F	-1.e5	track path length from its origin to ToF bar
cds	F	-1.e5	track path length from its origin to ToF calib. bar
beta	F	-1.e5	ToF calculated v/c for this track
LL[5]	F	0	Log-likelihood PID for different particle species

Table 6.10: Description of the `MIPPToFSummary` class. Var. type I = integer, F = float.

Summary of `MIPPDCkovSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
itrk	I	0	Track index
mirr	I	0	associated central DCkov mirror
ntdc[3][3]	I	0	Num. of TDCs of 3x3 grid about central mirror
tdc[3][3][10]	I	0	Up to 10 TDCs of 3x3 grid about central mirror
adc[3][3]	I	0	ADCs of 3x3 grid about central mirror
x	F	-1.e5	projected x-pos. of track at up. end of DCkov
y	F	-1.e5	projected y-pos. of track at up. end of DCkov
xp	F	-1.e5	projected dx/dz of track at up. end of DCkov
yp	F	-1.e5	projected dy/dz of track at up. end of DCkov
ds	F	-1.e5	path length of track inside DCkov
LL[5]	F	0	Log-likelihood PID for different particle species

Table 6.11: Description of the `MIPPDCkovSummary` class. Var. type I = integer, F = float.

Summary of `MIPPRICHSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
itrk	I	0	Track index
mirr	I	0	projected RICH mirror
nhits	I	0	Num. of PMT hits attributed to this track
rad	F	-1.e5	RICH ring radius
drad	F	-1.e5	RICH ring radius uncertainty
xr	F	-1.e5	x-center. of RICH ring
yr	F	-1.e5	y-center. of RICH ring
xt	F	-1.e5	projected x-pos of track at face of PMT box
yt	F	-1.e5	projected y-pos of track at face of PMT box
LL[5]	F	0	Log-likelihood PID for different particle species

Table 6.12: Description of the `MIPPRICHSummary` class. Var. type I = integer, F = float.

Summary of `MIPPCaloSummary` Class - Round 3			
Var. name	**Var. type**	**Default value**	**Description**
itrk	I	0	Track index
ext	F	-1.e5	projected x-pos of track at ECal
eyt	F	-1.e5	projected y-pos of track at ECal
exc	F	-1.e5	projected x-pos of track at ECal
eyc	F	-1.e5	projected y-pos of ECal cluster
eecal	F	-1.e5	reconstructed energy deposited in ECal
hxt	F	-1.e5	projected x-pos of track at HCal
hyt	F	-1.e5	projected y-pos of track at HCal
hxc	F	-1.e5	projected x-pos of track at HCal
hyc	F	-1.e5	projected y-pos of HCal cluster
ehcal	F	-1.e5	reconstructed energy deposited in HCal

Table 6.13: Description of the `MIPPCaloSummary` class. Var. type I = integer, F = float.

Chapter 7

Retrieving MIPP data from Enstore

MIPP data is stored on robot tapes in FCC. This chapter details how data can be extracted so that it can be worked on.

7.1 Data storage system documentation

The other sections in this chapter detail MIPP specific ways to get data. If you are interested in getting more documentation about data storage system, the following web sites will be useful.

- http://computing.fnal.gov/docs/products/enstore/

- http://www.dcache.org/

- http://hppc.fnal.gov/enstore/

7.2 Organization of MIPP Enstore area

MIPP home area in Enstore contains two actively used directories: `mippData` where raw data files are stored and `mippPostgres` where weekly database dumps are stored for backup.

`mippData` directory contains a 5-digit subdirectories, each containing 1000 runs beginning with the five digits. For example, run 13432 will reside in `mippData/00013`.

`mippPostgres` directory contains 8-digit subdirectories for the date of database dump of the form `YYYYMMDD`. All databases on the database server (mippsrv01) are stored as separate files. Offsite clusters may wish to mirror databases locally to speed up access to database information. See Postgres documentation on how to do this.

7.3 On e907ana computers

If you are looking for a recent run (a few days old), then the file is likely to be in either `/data/0` or `/data/1`. If you intend to keep the file on the cluster for a while, then copy it to one of the `/disks/X` directories. *Please do not copy files to* `/home` *partition!*

If the run you are looking for is not in `/data/0` or `/data/1`, then you will have to extract the file from Enstore. Before you proceed, make sure that `/usr/local/bin` is in your `PATH` environment variable.

You have two ways to extract data:

1. `ENSgetRun.pl --run <run> --dir <directory>`
   Example: `ENSgetRun.pl --run 13200 --dir .`

2. `encp /pnfs/e907/mippData/NNNNN/<data file> <directory>`
   Example: `encp /pnfs/e907/mippData/00013/mipp00013200.*.raw .`

The first option is a Perl script that retrieves all sub runs for a given run. The second option allows you to use wildcards, etc. For all intents and purposes, `encp` works just like `cp` when operated on PNFS file system. Again, please do not copy files to `/home` partition!

7.4 Through kerberized FTP

If you would rather get files directly, bypassing e907ana cluster (suppose you want to do processing off-site!), then kerberized FTP is presently the only way to get data files.

MIPP users' kerberos principals (user names) should be added to Enstore configuration to allow access to our data. If your principal is not added, file a helpdesk ticket. Contact David Lange or Andre Lebedev if you continue to have trouble.

Once you obtain a kerberos ticket (`kinit -f <user name>`), you should be able to get read only access by executing

```
> ftp fndca.fnal.gov 24127
Connected to stkendca3a.fnal.gov.
220 Kerberos FTP Door ready
334 ADAT must follow
GSSAPI accepted as authentication type
GSSAPI authentication succeeded
Name (fndca.fnal.gov:alebedev):  alebedev
200 User alebedev logged in
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

You should use your kerberos principal (user name) in order to get access. Once you are in, the following command will get subrun 0 of run 13200:

```
get mippData/00013/mipp00013200.0000.raw
```

Fndca FTP server accepts Java style wildcards, so to get all sub runs of run 13200 particular run, execute

```
mget mipp00013200.000..raw
```

7.4.1 Hints

1. *Log in without prompt*
   If you would like for ftp to log you on automatically, add this line to `/.netrc` file (create the file if you don't have one):
   ```
   machine fndca.fnal.gov login your-kerberos-principal
   ```

2. *Turn of confirmation for downloading*
   FTP `prompt` command toggles the switch for interactive promting during multiple file transfers.

3. *About Java wildcards*
   The website below should be helpful if you want to do more fancy tricks than using a dot as a wildcard symbol.

   ```
   http://java.sun.com/j2se/1.4.2/docs/api/java/util/
                              regex/Pattern.html
   ```

Chapter 8

MippDatabase: SQL database interface

8.1   Introduction

The MippDatabaseBase and MippDatabase classes provide an interface from MIPP code to an SQL database. The database implemented is PostGreSQL. This document explains how PostGreSQL is configured on MIPP computers, how to load the database with tables defined in XML, and how to retrieve this information from them in MIPP executables.

This section assumes familiarity with other software used by MIPP and described in this document, specifically SRT and root.

8.2   To-Do list

- Everything seems too slow (but it seems to be the PostGres software itself..)

- Make overriding the db with XML easier

8.3   Underlying SQL software

We have selected the PostGreSQL version of the SQL relational database standard. PostGreSQL is open source (as opposed to Oracle) and implements a larger fraction of the standard than mySQL.

The actual details of PostGreSQL are hidden from the user interface (and even much of the code described here). This will allow MIPP to switch to a different implementation of rational databases with minimal effort.

For our offline code, we are currently using the C interface that comes in the standard PostGreSQL distribution.

8.4 Setup for PostGreSQL

1. Download a recent PostGreSQL tar file from the postgres web site. Start at http://www.postgresql.org/mirrors-ftp.html. We are using version 7.4.5, however newer releases should also be fine.

2. `tar xvz  lange6/postgresql-7.4.5.tar.gz`

3. `cd postgresql-7.4.5`

4. See the INSTALL and configure files for more information. In order to make use of python scripts for DB access, you should add `--with-python` option when running `configure` script.

5. `./configure`

6. `gmake`

7. (as root) `gmake install`

8. Now you sure have PostGres installed in `/usr/local/pgsql`. The 'lib' subdirectory should contain libraries including libpq.

If you want to use your local installation to mirror the MIPP db or otherwise use your PostGreSQL to keep data locally you will in addition need to do

1. (as root) `mkdir /usr/local/pgsql/data`

2. (as root) `chown lange /usr/local/pgsql/data`

3. (Not as root) Initialize the data directory for your database

    `/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data`

52

4. The 'postmaster' process needs to be running when you want to access the database:

```
/usr/local/pgsql/bin/postmaster -D
      /usr/local/pgsql/data >& logfile &
```

5. To create a database, do `createdb <db-name>`. This only needs to be done once. For MIPP the following databases exist:

   - `cmap`: Connection maps
   - `calib`: Calibration constants
   - `runs`: Online database, contains run configuration and status information
   - `slowmon`: All slow controls monitoring information
   - `rich`: RICH electronics connection map, board status

In addition, to compile and run MIPP code that relies on the database, you will need to define the environment variable `PGSQL_DIR` to be the location where you installed PostGreSQL (by default this is `/usr/local/pgsql`) and you will need to add `PGSQL_DIR/lib` to your `LD_LIBRARY_PATH`. This should be done for you in the standard setup scripts on e907daq.

See also HTML documentation under the `pgsql/doc/html` directory for available database management commands.

8.4.1 PostGreSQL on e907anaX/e907daq

The PostGreSQL database is hosted on mippsrv01 and mippsrv02. mippsrv02 is a backup to mippsrv01. Use mippsrv01 to write new data to the db. The database host can be specified in the environment variable PGHOST.

We have two user accounts for accessing the database: `mippdbread` and `mippdbwrite`. Passwords have recently been sent to the mipp-priv mailing list. If not ask me (lange6@llnl.gov) or others what they are. To set the database user, simply do (if you are a tcsh user..)

```
setenv PGUSER mippdbread
```

As the names indicate, mippdbread is for processes that do not need to write information to the database, and mippdbwrite is for those that do. The `mippdbread` user can only query the database, and should the the default user for offline applications.

To most easily set up your account to access the MIPP database, create a `.pgpass` file that looks like

```
*:*:*:mippdbwrite:readerpassword
*:*:*:mippdbread:writerpassword
```

Be sure that the permissions for this file are 0600; otherwise it will be ignored by PostGreSQL.

8.4.2 Remote access

The database on mippsrv0n can currently be accessed remotely via TCP/IP. Your connection needs to be SSL encrypted. The following environment variable is also needed:
`PGHOST = mippsrv01.fnal.gov`
Offline code looks for `PGHOST` variable to be set. If it is not set explicitly, than the code defaults to `mippsrv01.fnal.gov`. Therefore, if you wish to use a mirror of the database, set the environment variable.

8.4.3 Database backup

The database is backed up regularly using a script in the Enstore package (mipp/online/Enstore in cvs). Archived database snapshots are found in Enstore at

```
/pnfs/e907/mippPostgres/<date>
```

and can be restored using:

```
pg_restore <file to restore from>
```

Note that the files stored in Enstore are gzipped, so they would need to be unzipped before the pg_restore command could be used. See PostGreSQL documentation for details.

8.5 MIPP specific code

The code for the MIPP database implementation currently lives in two packages: MippDatabaseBase and MippDatabase. These packages depend on the Conventions package and the MippXML package. The MippDatabaseBase package contains a generic (but simple in terms of the feature set implemented) SQL interface and the PostGreSQL implementation for that interface. This is designed to make it possible for MIPP to use a different SQL database in the future, with a minimal change in code. The MippDatabase package implements the specific tables needed for MIPP use as well as the XML interface.

8.6 Types of predefined tables

The MIPP database software predefines two types of tables that we believe are needed for general purpose use in MIPP software. These are a table based on run number and sub-run number ranges and a table based on time stamp. Tables based on these basic types are configurable either in code

- `MippDatabase/MdbTableRunSubRun`

- `MippDatabase/MdbTableOneRun`

- `MippDatabase/MdbTableTimestamp`

or via XML

- `MippDatabase/MdbTableRunSubRunXML`

- `MippDatabase/MdbTableOneRunXML`

- `MippDatabase/MdbTableTimestampXML`

We expect that the most common usage will be by XML, so we document that interface below.

Both tables are based on a set of user defined columns. These columns are augmented by columns which define the validity range, either by run number or by time stamp, as well as defining the time at which the database entry was loaded into the database.

The "RunSubRun" table adds columns for the min and max run number ("runNumMin" and "runNumMax"), min and max sub-run number ("subRunMin" and "subRunMax"), and load time ("loadTimeGMT"). The "Timestamp" table adds columns for the min and max validity time ("tsMin" and "tsMax") and the load time ("loadTimeGMT"). The "OneRun" table adds columns for the run number ("runNum") and load time ("loadTimeGMT").

There is one caveat for the Timestamp table. Intervals should be defined as not to overlap. To avoid actually doing the database query each time, the result of each query is cached and then used until it is no longer valid.

8.7 XML interface

```
<dbTable TableName="testDBTable"
        TableType="RunSubRun"
        creationDate="031030"
        maintainer="lange6@llnl.gov"
        WriteXMLToDB="False"
        OverrideDBWithXML="False"
        XMLThenDB=''False''
        DBName="test">

<dbTableColumn> Crate INTEGER </dbTableColumn>
<dbTableColumn> Module INTEGER 10 </dbTableColumn>

<xmlfile>
MippDatabase/xml/test/dbentry1.xml
</xmlfile>

</dbTable>
```

The tag `xmlfile` is one of the basic tags defined in the MippXML class. In this file we define a table with name "testDBTable" that contains two columns, "Crate" and "Module". These columns are of an integer type. The optional third argument specifies the maximum number of entries in the column or the maximum length of the character string in VARCHAR columns. For PostGreSQL, the maximum number of entries is not really

used except for being 1 or greater than one. If the default is specified (=1), then arrays of numbers can not be used for that column. On the contrary, if a number greater than one is specified, then arrays of any length are allowed. Arrays of VARCHARs are not currently supported in the XML interface.

There are three database types, either "RunSubRun", "Timestamp", or "OneRun".

The flag "WriteXMLToDB" controls whether or not the data contained in the XML file is written into the database. This should only be set to "True" if you are loading new data into the database, and generally should be set to "False". If the flag "OverrideDBWithXML" is set to true, the values found in the database for this table are overridden by those in XML. Similarly, if the flag "XMLThenDB" is set, then values read from the XML are used as the default, but if no valid value is found, the underlying database table is used instead. Note that "WriteXMLToDB", "OverrideDBWithXML", and "XMLThenDB" are mutually exclusive: your code will core dump if both are set to "True".

The file dbentry1.xml contains a "dbTableEntry", which is the set of rows that correspond to one database entry.

```
<dbTableEntry TableName="testDBTable"
              TableType="RunSubRun"
              runNumMin="1"
              runNumMax="1000"
              creator="lange6@llnl.gov">

<dbRow> 1, 1 </dbRow>
<dbRow> 1, 2 </dbRow>
<dbRow> 1, 3 </dbRow>
<dbRow> 1, 4 </dbRow>
<dbRow> 1, 5 </dbRow>
<dbRow> 2, 7 </dbRow>
<dbRow> 2, 4 </dbRow>
<dbRow> 2, 3 </dbRow>
<dbRow> 2, 2 </dbRow>
<dbRow> 2, 1 </dbRow>

</dbTableEntry>
```

In this case there are ten rows for this entry, each defining a value for the Crate and the Module column. The order of the "dbRow" tag is required to be the same as the order in which columns were declared in the "dbTable" declaration.

This entry is valid for runs 1-1000. One may also specify a sub-run range. In that case, the "runNumMin" and "runNumMax" must be equal.

In addition, arrays of numbers can be specified via

```
<dbRow> 2 2 3 10, 1 </dbRow>
```

where the "," separates the columns in the table, and spaces separate the list of numbers within a column.

8.8   Special characters

I do not have an authoritative list of special characters in SQL, specifically PostGreSQL. However, the following should be avoided when defining table names, column names, and in VARCHAR entries: "$", "()", "[]", ",", ";", ":", "*", ".".

8.9   Concrete example: Ckov cable map

There are two pieces needed to implement and use a cable map in the database. First, the table must be defined and loaded. Second, the table must be retrieved.

8.9.1   Define and load a table into the database

The CkovReco package contains a simple executable to load a table and to print out its content in the LoadCkovDB.cc executable:

```
#include <iostream>
#include <string>
#include <vector>
#include "MippXML/MXML.h"
#include "MippXML/MXMLStack.h"
#include "MippDatabase/xml/MXMLdbTableBuilder.h"
#include "MippDatabase/MdbTableDict.h"
```

```
#include "MippDatabase/MdbDatabase.h"
#include "MippDatabase/MdbTableRunSubRun.h"
#include "MippDatabase/MdbTableTimestamp.h"
#include <fstream>
//....................................................

int main(int argc, char** argv)
{
  MdbDatabase *dbFact=MdbDatabase::Instance();
  MXML::Initialize();
  MXML::ReadFile(argv[1]);

  MdbTableDict *tableDict=MdbTableDict::Instance();

  std::vector<std::string> tables;
  tables.push_back("CkovReadout");

  for ( unsigned int i=0; i< tables.size(); ++i) {


    MdbTableRunSubRun *table=
      tableDict->GetTableRSR(tables[i]);

    if ( table ) {

      table->CacheAllDBRows();
      table->printTable();
    }
    else{
      std::cout << "Table " << tables[i] << " missing ??\n";
    }
  }

  return 0;
}
```

The GNUmakefile of CkovReco looks like

```
ROOTCINT     :=
```

```
SUBDIRS      :=
BINS         := LoadCkovDB
SIMPLEBINS   := LoadCkovDB
LIB_TYPE     := shared
LIB          := lib$(PACKAGE)
LIBCXXFILES := $(wildcard *.cxx)

NODEP_LIBS +=

LOADLIBES += \
-lMippDatabaseXML \
-lMippDatabase \
-lMippDatabaseBase \
-lMippXML
#########################################################
include SoftRelTools/standard.mk
include SoftRelTools/arch_spec_root.mk
include SoftRelTools/arch_spec_xercesc.mk
include SoftRelTools/arch_spec_pgsql.mk
```

LoadCkovDB is run as

```
LoadCkovDB < xml-file >
```

An example xml file is

```
CkovReco/xml/loadCkovReadout.xml,
```

which looks like

```
<xmlfile>
    CkovReco/xml/CkovReadout.xml
</xmlfile>
```

where CkovReco/xml/CkovReadout.xml defines the format of the cable map table

```
<dbTable TableName="CkovReadout"
         TableType="RunSubRun"
         creationDate="040416"
```

```
            maintainer="lange6@llnl.gov"
            WriteXMLToDB="False"
            OverrideDBWithXML="False"
            XMLThenDB="False"
            DBName="test">

<dbTableColumn> Mirror INTEGER </dbTableColumn>
<dbTableColumn> ADCCrate INTEGER </dbTableColumn>
<dbTableColumn> ADCSlot INTEGER </dbTableColumn>
<dbTableColumn> ADCChannel INTEGER </dbTableColumn>
<dbTableColumn> SpiderBox INTEGER </dbTableColumn>
<dbTableColumn> SpiderBoxChan INTEGER </dbTableColumn>
<dbTableColumn> TDCCrate INTEGER </dbTableColumn>
<dbTableColumn> TDCSlot INTEGER </dbTableColumn>
<dbTableColumn> TDCFirst INTEGER </dbTableColumn>

<xmlfile>CkovReco/maps/CkovReadout_v1.xml </xmlfile>
</dbTable>
```

The important items in the header are

- *TableName*

- *TableType*

- *WriteXMLToDB* True if you want to write the data defined in XML to the database, false otherwise

- *OverrideDBWithXML* True if you want to use the data defined in XML instead of data stored in the database.

The actual data is defined in CkovReadout_v1.xml

```
<dbTableEntry TableName="CkovReadout"
              TableType="RunSubRun"
              runNumMin="1"
              runNumMax="10000000"
              creator = "lange6@llnl.gov">
<dbRow> 1,  3,  3,  1,  1,  1,  3,  8,  -1  </dbRow>
```

where the dbRow gives an entry for each column defined in the "dbTable" definition.

8.9.2 Retrieving data from the database in MIPP executables

The Ckov map defined in the section above is used in the raw2root executable, in the R2REvent class. The database is currently too slow to be queried every event, so a few variables are defined to store the relevant information, in this case, the mirror number correspondence to ADC and TDC channel.

```
std::map<int,int> fCkovADCHash;
std::map<int,int> fCkovTDCHash;
```

These maps are filled in the DoCkov method:

```
static int first=1;

if ( first == 1 ) {
  first=0;
  MdbDatabase *dbFact=MdbDatabase::Instance();
  MXML::Initialize();
// make sure that WriteXMLToDB is False !
  MXML::ReadFile("CkovReco/xml/loadCkovReadout.xml");

  MdbTableDict *tableDict=MdbTableDict::Instance();
  std::vector<std::string> cols;

// query the table and get the entry for the current run
// the last argument means that only the most recently stored
// set of entries will be retrieved.  Otherwise, you'll get all
// entries, but entries will be sorted according to load time
  MdbAbsRelDBTable *mapTable =
    tableDict->GetTableRSR("CkovReadout")->Select(
    cols,this->fHeader.Run(),-1,0,0,0,''Mirror'' );

  std::string ADCCrate("ADCCrate");
  std::string ADCSlot("ADCSlot");
  std::string ADCChannel("ADCChannel");
  std::string Mirror("Mirror");
  std::string SpiderBoxChan("SpiderBoxChan");
  std::string TDCFirst("TDCFirst");
```

```
    std::string TDCCrate("TDCCrate");
    std::string TDCSlot("TDCSlot");

// Loop over the rows retrieved from the database.
    for ( unsigned int i=0;
        i<mapTable->NumberOfCachedRows(); ++i) {

// 1000*create + 100*slot + channel
    fCkovADCHash[
        1000*mapTable->getCachedTableDataPtr(
                ADCCrate,i)->valInt()[0]
        +100*mapTable->getCachedTableDataPtr(
                ADCSlot,i)->valInt()[0]
        +mapTable->getCachedTableDataPtr(
                ADCChannel,i)->valInt()[0]  ] =
mapTable->getCachedTableDataPtr(
                Mirror,i)->valInt()[0];

// 1000*create + 100*slot + channel
    int spidChannel=mapTable->getCachedTableDataPtr(
                SpiderBoxChan,i)->valInt()[0];
    fCkovTDCHash[ 1000*mapTable->getCachedTableDataPtr(
                TDCCrate,i)->valInt()[0]
    +100*mapTable->getCachedTableDataPtr(
                TDCSlot,i)->valInt()[0]
    +    spidChannel ] =
mapTable->getCachedTableDataPtr(Mirror,i)->valInt()[0];

    }
  }
```

In the code above "most recent" means that you'll get the most recent row corresponding to each unique entry in the column "Mirror".

8.9.3 Adding data to tables in C++ code

The other common application of the database will be to add data in C++ applications, such as the high voltage monitoring. Here is an example for

how to do that for the CkovReadout table defined above

```
int main(int argc, char** argv) {
  MdbDatabase *dbFact=MdbDatabase::Instance();
  MXML::Initialize();
  MXML::ReadFile(argv[1]);

  MdbTableDict *tableDict=MdbTableDict::Instance();

  MdbTableRunSubRun *table=tableDict->GetTableRSR(
        std::string(''CkovReadout''));

  MdbDatabase *dbFact=MdbDatabase::Instance();

  std::vector<MdbAbsRelDBVar*> vars;
  vars.push_back(dbFact->dbVar(int(100))); //mirror
  vars.push_back(dbFact->dbVar(int(4))); //ADCCrate
  vars.push_back(dbFact->dbVar(int(14))); //ADCSlot
  vars.push_back(dbFact->dbVar(int(15))); //ADCChannel
  vars.push_back(dbFact->dbVar(int(7))); //SpiderBox
  vars.push_back(dbFact->dbVar(int(1))); //SpiderBoxChan
  vars.push_back(dbFact->dbVar(int(4))); //TDCCrate
  vars.push_back(dbFact->dbVar(int(13)); //TDCSlot
  vars.push_back(dbFact->dbVar(int(1))); //TDCFirst

// runMin, subrunMin, runMax, subrunMax
  table->InsertRow(vars,1,-1,1000,-1,-1);

//clean up
  for ( unsigned i=0; i<vars.size(); i++ ) delete vars[i];

  return 0;
}
```

8.10 MippDatabase Classes

8.10.1 SQL basics

This section to be developed in time.

The basic SQL commands, as well as the pgsql interface that we are using, are based on character strings. The basic layer on top of the pgsql (in MippDatabaseBase) is to convert strings to and from the actual variable types (floats, etc). The interface used by MIPP tables is then formed from this extra layer of code. In this way, we are exposed primarily to type specific interfaces. However, in some cases, the underlying character string interface means that some errors will only be caught at runtime, in particular in parsing values from XML files. We have worked to make the interface so that mistakes result in compile time errors, but this is not always possible.

8.10.2 Initialization

To initialize the database and read in the XML tables

```
MdbDatabase *dbFact=MdbDatabase::Instance();
MXML::Initialize();
MXML::ReadFile(argv[1]);
```

is needed in your executable. In this example, argv is the name of a top level XML file.

See MippDatabase/xml/test/test/testDbXML.cc for an example (MippDatabase/xml/test/test.xml is an example input for this executable).

All tables needed by your job must be either defined in C++ code or read in via the XML interface. This is so that the structure of the data is known to the database interface.

8.10.3 Retrieving a table

To get a table read in via the XML interface:

```
MdbTableDict *tableDict=MdbTableDict::Instance();
MdbTableRunSubRun *table=
        tableDict->GetTableRSR("testDBTable");
```

the "table" pointer will be null if the table is not found in the dictionary. This means that it was not read in via the XML interface.

8.10.4 MdbDatabase class

The MdbDatabase class allows the specific implementation of the SQL database to be hidden for the rest of MIPP code. Although only a PostGreSQL based implementation is foreseen, this class would allow for other implementations to be used instead, with no changes to user code.

The MdbDatabase class is found in the MippDatabase package and is used to create other database objects, as explained below.

8.10.5 Database variables

The base type for variables to be stored (or retrieved from) the database is MdbAbsRelDBVar, which is found in the MippDatabaseBase package. The implementation for PostGreSQL is MdbPostGresVar, found in the same package.

Four variable types are currently implemented

1. Integer (MdbAbsRelDBVar::INTEGER)

2. Float (MdbAbsRelDBVar::FLOAT)

3. String (MdbAbsRelDBVar::VARCHAR)

4. Timestamp (MdbAbsRelDBVar::TIMESTAMP)

Arrays of each of these types are supported. Timestamps are implemented in TTimeStamp, which is a part of ROOT. Times are assumed to be in GMT.

The interface to create a database variable object is a bit clumsy:

```
// do this once
MdbDatabase *dbFact=MdbDatabase::Instance();

// Remeber to delete five later
MdbAbsRelDBVar *five=dbFact->dbVar(int(5));
```

The current list of available methods to create database variables is

```
MdbAbsRelDBVar* dbVar( int val);
MdbAbsRelDBVar* dbVar( std::vector<int> val);

MdbAbsRelDBVar* dbVar( float val);
```

```
MdbAbsRelDBVar* dbVar( std::vector<float> val);

MdbAbsRelDBVar* dbVar( TTimeStamp val);
MdbAbsRelDBVar* dbVar( std::vector<TTimeStamp> val);

MdbAbsRelDBVar* dbVar( std::string val);
MdbAbsRelDBVar* dbVar( MdbAbsRelDBVar::RelDBVarType type,
                       std::string val);
```

The last of these allows you to enter directly the string to be passed to the database. We suggest that this be used by experts only.

To retrieve values from the database variables, the following methods are provided

```
std::vector<float> valFloat() const ;
std::vector<int> valInt() const ;
std::vector<std::string> valStr() const ;
std::vector<TTimeStamp> valCTS() const ;
```

Note that if the database variable is of a different type, then an empty vector will be returned.

8.10.6   Selecting a subset of rows

The database implementation allows for quite a general select (from a single table) query. We describe here the basic ones. For the RunSubRun-based table, this is to search for the conditions valid for a given run and sub-run, and for the Timestamp-based table to search for the conditions valid for a given time stamp.

The method for the RunSubRun table is

```
MdbAbsRelDBTable* Select(
                   std::vector<std::string> cols,
          int run, int subRun=-1,
   TTimeStamp *loadTime=0,
   MdbAbsRelDBConstraint *reqs=0,
   MdbAbsRelDBOrder *orderBy=0,
   std::string uniqKey='''');
```

67

There are two required arguments. The first one is the list of columns you would like returned. If the vector is empty, then all columns will be returned. This is the recommended usage. Second, the run number you are interested in. The other arguments are optional. The third argument is the sub-run number, used only if the validity period of the conditions does not span run numbers. Fourth, if you would like to ignore recently loaded conditions, you can specify a time stamp. Rows in loaded into the database before this time will be ignored. Next, is any additional constraints you would like to use. This option is not documented yet. Sixth, is an option to specify the order in which the rows should be returned. The default order is in the reserve order in which rows were loaded into the database. The orderBy option lets you further refine that order. This option is not yet documented. The last argument allows you to specify if you would like to have only the most recently loaded conditions returned by your query, keyed by the column specified in the argument. If nothing is specified, then all valid rows will be returned. If a column name is specified then exactly one row (the most recent one) per value of the uniqKey column will be returned. So for example, if you have loaded various sets of row, each set having a row per detector channel, you can ask for the most recent row for each detector channel by specifying the appropriate row for uniqKey.

The returned object is a "MdbAbsRelDBTable", the code for which lives in MippDatabaseBase. This class is described below.

An example Select (to get the rows corresponding to run 100) is:

```
MdbTableRunSubRun *table=
   tableDict->GetTableRSR("testDBTable");

if ( table ) {
  table->printTable();

  // lets also do a simple search on this table.
  std::vector<std::string> cols;
  MdbAbsRelDBTable *resTable = table->Select(cols,100);
}
```

To avoid memory leaks, resTable should be deleted.

8.10.7 Creating XML files from a database table

It is simple to write out a database table in the XML format for future use. For example, if you wish to test your algorithm changing only one number in a database table, you could write out the database table, make your change, and then override the information in the database with your changed XML file for testing. Once your tests are successful, the same XML file (with a minor change to the "WriteXMLToDB" parameter described above) can be used to load the database with the new conditions.

The syntax for writing out to the XML format is

```
// get the table we want
MdbTableRunSubRun *table=
    tableDict->GetTableRSR("testDBTable");
// make a stream for the output
std::ofstream fp("testOutput.runsubrun");
// make the XML file
table->MakeXMLFile(fp,true,0);
```

The three arguments to the MakeXMLFile method are:

1. (std::ofstream, required) The ofstream to write to

2. (bool, default = false) Control if the full table is to be read from the database. If true, then only the cache of the table is used. If false, then all data is printed.

3. (MdbAbsRelDBTable*, default = 0) The result of a Select that should be printed instead of the nominal table in the MdbTableRunSubRun object. You may be interested in printing the results of a "select" query instead of the full table. The query will return a MdbAbsRelDBTable object with your selected rows.

8.10.8 The MdbAbsRelDBTable class

An object of type MdbAbsRelDBTable is returned by queries to the database. We document a subset of the functionality of this class here. The actual implementation of this type for PostGreSQL is the MdbPostGresTable class.

To find the number of data rows and columns in the data, use these methods:

69

```
virtual unsigned int NumberOfCachedRows()
std::string ColumnName(unsigned int i)
virtual unsigned int NumberOfColumns()
```

We expect that the only time MIPP users will use the MdbAbsRelDBTable type is as a result of selecting a subset of information from a RunSubRun or Timestamp table. In this case, the table "cache" contains the the result of the query. So to find out the number of rows returned by your query, use the `NumberOfCachedRows` method.

To retrieve the data from the the cache, use the getCachedTableData or getCachedTableDataPtr methods:

```
virtual MdbAbsRelDBVar*
    getCachedTableData(std::string columnName,
                       unsigned int rowNum);
virtual MdbAbsRelDBVar*
    getCachedTableDataPtr(std::string columnName,
                          unsigned int rowNum);
```

The getCachedTableData method news the returned variable and gives ownership of the object to the calling method. In most cases this is unnecessary.

You can also print the cached table information in several ways

```
void printTable(std::ostream& o = std::cout);
void printTableHtml(std::ostream& o = std::cout);
void printTableTex(std::ostream& o = std::cout);
```

Further methods are documented in the header file. These tend to be low level methods that should not in general be needed outside of the core database code.

70

Chapter 9

Residual corrections to TPC hits: TPCResCor

9.1 What is TPCResCor?

The reconstruction of the location of an ionization cloud from the information recorded on the TPC pad plane is not an easy business. The electrons must be tracked backwards through a rather inhomogeneous B field with some assumption being made about the terminal velocity v_{Drift}. Additionally, there may be alignment uncertainties and possibly shaping time differences from padrow to padrow. While we make every effort to calibrate and understand each of these effects, there may continue to be systematic differences between where hits are reconstructed and where we think they should be based on a complete fit to the track. The package `TPCResCor` is designed to facilitate the calculation and application of these residual offsets to the reconstructed TPC hits.

9.2 Method

The offsets used by TPCResCor are arrived at iteratively. A track fitter (it could be any fitter) computes the differences between the fitted track position and each TPC hit as a function of the TPC hit location in x and y and pad row address. These differences are accumulated and at the end of the run the average offset in both x and y are computed as functions of the hit position (x,y) and pad row address. To avoid biases from tails, only the center part

of the distribution (out to the FWHM) are used in the averages. The x, y binning is configurable, but a typical binning is to use 8 bins in each of x and y with bin edges located at $x = (-50, -35, -20, -10, 0, 10, 20, 35, 50)$ cm, and $y = (-25, -15, -5, 0, 5, 15, 25, 35, 55)$ cm. The binning compromises statistics, granularity, and simplicity. When offsets are requested for a given TPC hit, the x and y offsets used are interpolated from the tabulated results for that pad row.

The `TPCResCor` package is not intended to replace a good understanding of the TPC hit reconstruction, but is rather intended to make the last small changes to the hit positions after our best knowledge has been applied.

9.3 Using TPCResCor

Use of the package is illustrated in this example code fragment:

```
#include "TPCResCor/TPCResCorMap.h"
static TPCResCorMap* gsTPCResCor = 0;

void Module::NewRun(int run, int subrun)
{
  if (gsTPCResCor) { delete gsTPCResCor; gsTPCResCor = 0; }
  if (gsTPCResCor==0) {
    gsTPCResCor = new TPCResCorMap(run)
  }
}

void Module::Reco(EDMEventHandle& evt)
{
  double hitx;    // TPC hit x position
  double hity;    // TPC hit y position
  double hitz;    // TPC hit z position
  int    ipadrow; // TPC hit pad row number

  // Load uncorrected hit position into hitx, hity, hitz, and
  // ipadrow...

  // Update the x/y position using the map
  gsTPCResCorMap->AdjustHit(&hitx,&hity,ipadrow);
}
```

The code creates a new map at the "new run" boundary. By default, the code will go to the database for the map which is most appropriate for the specified run number. Alternately, one can load the table from a text file. In that case, you would instantiate the TPCResCorMap object using two arguments, the run number, and the name of the text file you want to load: `new TPCResCorMap(1000,"myfile.txt")`. We plan to apply the adjustments at the time that the TPC hits are created and stored, so there should be only one place (in `TPCRecoJP`) where these offsets need be applied. Therefore, you may assume that TPC hits have been adjusted for you and that any further adjustments would be an over correction. If you want to be sure the adjustments are being applied at the TPC reconstruction stage in your job check the XML configuration parameters for the `TPCRecoJP` modules.

9.4 Updating TPCResCor

As mentioned in the introduction, the method for producing the offset tables is by an examination of track residuals. Global track fitters (eg. SP-Fit, SPTrk, and Kalman) can update the tables by having their residuals accumulated and analyzed. The class that aids this process is called `TPCResCorMaker`. Here is some code:

```
#include "TPCResCor/TPCResCorMap.h"
#include "TPCResCor/TPCResCorMaker.h"
static TPCResCorMap*   gsTPCResCorMap   = 0;
static TPCResCorMaker* gsTPCResCorMaker = 0;

void Module::NewRun(int run, int subrun)
{
  // Initialize the map and the makers
  if (gsTPCResCorMap)   delete gsTPCResCorMap;
  if (gsTPCResCorMaker) delete gsTPCResCorMaker;
  gsTPCResCorMap   = new TPCResCorMap(run);
  gsTPCResCorMaker = new TPCResCorMaker(gsTPCResCorMap);
}

void Module::Reco(EDMEventHandle& evt)
{
```

```
  for (i=0; i<nHitTpc; ++i) {
    double xtpc;    // X position of a TPC hit used in the track fit
    double ytpc;    // Y position of a TPC hit used in the track fit
    int    ipadrow; // Pad row hit was recorded on

    // For each TPC hit i, compute the track position and find the
    // residual. Assume in the following that xtpc,ytpc are the
    // TPC hit position for a hit located at ipadrow and xtrk,ytrk are
    // the best-fit track positions

    // Ask the maker to record this residual measurement
    gsTPCResCorMaker->AddResidual(xtpc, xtrk, ytpc, ytrk, ipadrow);
  }
}

void Module::EndRun(int run, int subrun)
{
  // Ask the maker to compute the best residual offsets and update the
  // map it was instantiated with
  gsTPCResCorMaker->Finalize();

  // Write the updated map to the database
  gsTPCResCorMap->WriteToDB();
}
```

The above code builds two classes. The first, `TPCResCorMap`, is the map of offsets that were used to adjust the TPC hits. The map is recalled by run number. The second, `TPCRecCorMaker`, is a helper class which performs the analysis of the residuals and updates the map for remaining differences. After being initialized each residual to TPC hits is logged by the maker. At the end of the run, the residual distributions are analyzed, fit, and updates to the residual map are pushed to the database. Alternately, the results can be pushed to a flat text file using `WriteTableToFile("myfile.txt")`.

9.5   Database Tables

The residual corrections are stored in two sets of tables defined by the `TPCResCorDB.xml` file. The first, `TPCResForFormat` records the parameters

used to generate the tables and contains information about the number of x bins, number of y bins, the number of pad rows used in the table. It also holds an array which mark the edges of the x bins and y bins. In principle, the format can be different for different run numbers which may aid in future reprocessing of the data should the format change. The second table is significantly larger and holds the actual residual data. This table, `TPCResCorData` holds a flag marking the correction as either a correction for x or y coordinate, the x, y bin numbers, and a list of offsets for each of the pad rows (typically an array of 128 numbers). To start the process of generating these tables, the database must be primed. A utility is provided for this in the `TPCResCor/test` directory:

```
mkseedfile --read tpc-resid-map-seed.txt --write db
```

which reads an text file defining the table format, sets the offsets all to zero, and loads the zeroed table to the database. The run number attached to this empty map is zero by default but can be set using the `--run` option. When processing a run, care should be taken to understand the starting point for the iterations. For example, if the map is initialized based on a run number, the starting point may be the current best map for the nearest run number. I suggest seeding the map from the zeroed text file `tpc-resid-map-seed.txt` on the first iteration rather than seeding the results from a previous run number. This can be accomplished from the `TPCResCor` package using:

```
> gmake tbin
> mkseedfile --run 123450 --read tpc-resid-map-seed.txt --write db
```

While seeding the iterations from the result of the previous run's results might produce a faster convergence, it introduces a complicated dependency of the reconstruction performance on the order in which runs were processed possibly resulting in a smoothing of discontinuous changes in running conditions.